2 LIFE CYCLES

Angelina Samaroo

INTRODUCTION

In the previous chapter, we looked at testing as a concept – what it is and why we should do it. In this chapter we will look at testing as part of overall software development. Clearly testing does not take place in isolation; there must be a product first!

We will refer to work-products and products. A work-product is an intermediate deliverable required to create the final product. Work-products can be documentation or code. The code and associated documentation will become the product when the system is declared ready for release. In software development, work-products are generally created in a series of defined stages, from capturing a customer requirement, to creating the system, to delivering the system. These stages are usually shown as steps within a software development life cycle.

In this chapter we will look at two life-cycle models – sequential and iterative. For each one, the testing process will be described, and the objectives at each stage of testing explained.

Finally, we will look at the different types of testing that can take place throughout the development life cycle.

Learning objectives

The learning objectives for each section are as follows. Each section has been categorised as K2 overall, but individual K1 elements are shown where applicable.

Software development models (K2)

- Understanding of the relationship between development, test activities and work-products in the development life cycle, giving examples based on project and product characteristics and context.
- Recognition that software development models must be adapted to the context of project and product characteristics. (K1)
- Recall of reasons for different levels of testing, and characteristics of good testing in any life-cycle model. (K1)

Test levels (K2)

- Be able to compare the different levels of testing, considering for each the:
  - Major objectives.
  - Typical objectives of testing.
  - Typical targets of testing (e.g. functional or structural) and related work-products.
  - People who test.
  - Types of defects and failures to be identified.

Test types (K2)

- Comparison of the four requirement types (functional, non-functional, structural and change-related) by example.
- Recognition that functional and structural tests can occur at any level. (K1)
- Identification and description of non-functional test types based on non-functional requirements.
- Identification and description of test types based on the analysis of a software system's structure or architecture.
- Explanation of the purpose of confirmation and regression testing.

Maintenance testing (K2)

- Recognition of differences between testing existing systems and new systems, considering:
  - Test types
  - Triggers for testing
  - Amount of testing
- Recall of reasons for maintenance testing (K1):
  - Modification
  - Migration
  - Retirement
- Description of the role of regression testing and impact analysis in maintenance testing.

Self-assessment questions
The following questions have been designed to assess the reader's current knowledge of this topic. The answers are provided at the end of the chapter.

Question SA1 (K2)
Which of the following is true about the V-model?

a. It has the same steps as the waterfall model for software development.
b. It is referred to as a cyclical model for software development.
c. It enables the production of a working version of the system as early as possible.
d. It enables test planning to start as early as possible.

Question SA2 (K2)
Which of the following is true of iterative development?

a. It uses fully defined specifications from the start.
b. It involves the users in the testing throughout.
c. Changes to the system do not need to be formally recorded.
d. It is not suitable for developing websites.

Question SA3 (K1)
Which of the following is in the correct order (typically)?

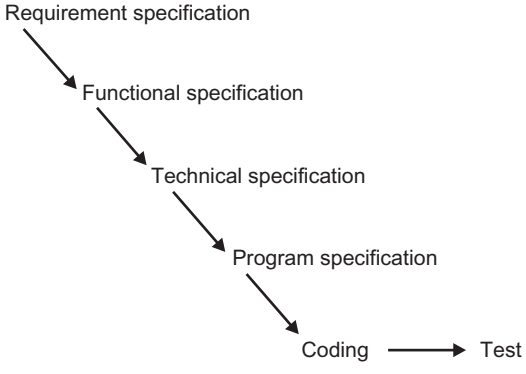
a. Unit testing, system testing, acceptance testing, maintenance testing.
b. System testing, unit testing, acceptance testing, maintenance testing.
c. Acceptance testing, system testing, maintenance testing, unit testing.
d. Unit testing, maintenance testing, system testing, acceptance testing.

SOFTWARE DEVELOPMENT MODELS

A development life cycle for a software product involves capturing the initial requirements from the customer, expanding on these to provide the detail required for code production, writing the code and testing the product, ready for release.

A simple development model is shown in Figure 2.1. This is known traditionally as the waterfall model.

Figure 2.1 Waterfall model



The waterfall model in Figure 2.1 shows the steps in sequence where the customer requirements are progressively refined to the point where coding can take place. This type of model is often referred to as a linear or sequential model. Each work-product or activity is completed before moving on to the next.

In the waterfall model, testing is carried out once the code has been fully developed. Once this is completed, a decision can be made on whether the product can be released into the live environment.

This model for development shows how a fully tested product can be created, but it has a significant drawback: what happens if the product fails the tests? Let us look at a simple case study.

CASE STUDY – DEVELOPMENT PROCESS

In a factory environment producing rivets for an aircraft fuselage, checks are made by operators to assess the rivets on a conveyor belt. This assessment may reveal a percentage of the rivets to be defective. Usually this percentage is small, and does not result in the whole batch of rivets being rejected. Therefore the bulk of the product can be released.

Consider now the same aircraft, but the product is the software controlling the display provided for the aircrew. If, at the point of testing, too many defects are found, what happens next? Can we release just parts of the system?

In the waterfall model, the testing at the end serves as a quality check. The product can be accepted or rejected at this point. As we saw in the case of rivet production, a single point of quality checking may be acceptable, assuming that most rivets pass the quality check.

In software development, however, it is unlikely that we can simply reject the parts of the system found to be defective, and release the rest. The nature of software functionality is such that removal of software is often not a clean-cut activity – this action could well cause other areas to function incorrectly. It may even cause the system to become unusable.

In addition, we may not be able to choose not to deliver anything at all. The commercial and financial effects of this course of action could be substantial.

What is needed is a process that assures quality throughout the development life cycle. At every stage, a check should be made that the work-product for that stage meets its objectives. This is a key point, work-product evaluation taking place at the point where the product has been declared complete by its creator. If the work-product passes its evaluation (test), we can progress to the next stage in confidence. In addition, finding problems at the point of creation should make fixing any problems cheaper than fixing them at a later stage. This is the cost escalation model, described in Chapter 1.

The checks throughout the life cycle include verification and validation.

Verification – checks that the work-product meets the requirements set out for it. An example of this would be to ensure that a website being built follows the guidelines for making websites usable by as many people as possible. Verification helps to ensure that we are building the product in the right way.

Validation – changes the focus of work-product evaluation to evaluation against user needs. This means ensuring that the behaviour of the work-product matches the customer needs as defined for the project. For example, for the same website above, the guidelines may have been written with people familiar with websites in mind. It may be that this website is also intended for novice users. Validation would include these users checking that they too can use the website easily. Validation helps to ensure that we are building the right product as far as the users are concerned.

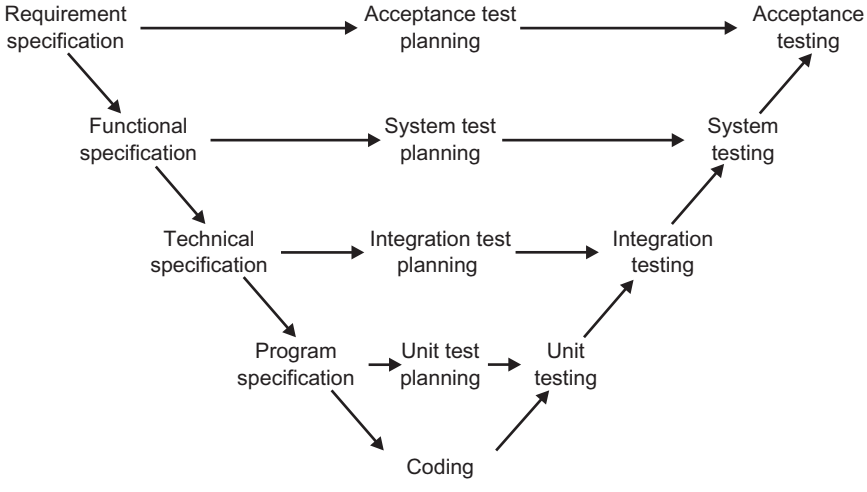
There are two types of development model that facilitate early work-product evaluation.

The first is an extension to the waterfall model, known as the V-model. The second is a cyclical model, where the coding stage often begins once the initial user needs have been captured. Cyclical models are often referred to as iterative models.

We will consider first the V-model.

V-model (sequential development model)
There are many variants of the V-model. One of these is shown in Figure 2.2.

Figure 2.2 V-model for software development



As for the waterfall model, the left-hand side of the model focuses on elaborating the initial requirements, providing successively more technical detail as the development progresses. In the model shown, these are:

- Requirement specification – capturing of user needs.
- Functional specification – definition of functions required to meet user needs.
- Technical specification – technical design of functions identified in the functional specification.
- Program specification – detailed design of each module or unit to be built to meet required functionality.

These specifications could be reviewed to check for the following:

- Conformance to the previous work-product (so in the case of the functional specification, verification would include a check against the requirement specification).
- That there is sufficient detail for the subsequent work-product to be built correctly (again, for the functional specification, this would include a check that there is sufficient information in order to create the technical specification).
- That it is testable – is the detail provided sufficient for testing the work-product?

Formal methods for reviewing documents are discussed in Chapter 3.

The middle of the V-model shows that planning for testing should start with each work-product. Thus, using the requirement specification as an example, acceptance testing would be planned for, right at the start of the development. Test planning is discussed in more detail in Chapter 5.

The right-hand side focuses on the testing activities. For each work-product, a testing activity is identified. These are shown in Figure 2.2:

- Testing against the requirement specification takes place at the acceptance testing stage.
- Testing against the functional specification takes place at the system testing stage.
- Testing against the technical specification takes place at the integration testing stage.
- Testing against the program specification takes place at the unit testing stage.

This allows testing to be concentrated on the detail provided in each work-product, so that defects can be identified as early as possible in the life cycle, when the work-product has been created. The different stages of testing are discussed later.

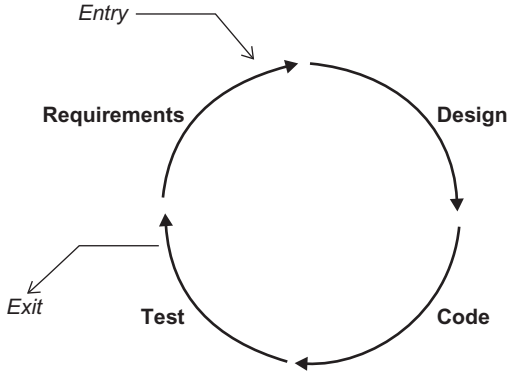
Remembering that each stage must be completed before the next one can be started, this approach to software development pushes validation of the system

by the user representatives right to the end of the life cycle. If the customer needs were not captured accurately in the requirement specification, or if they change, then these issues may not be uncovered until the user testing is carried out. As we saw in Chapter 1, fixing problems at this stage could be very costly; in addition, it is possible that the project could be cancelled altogether.

Iterative–incremental development models

Let us now look at a different model for software development – iterative development. This is one where the requirements do not need to be fully defined before coding can start. Instead, a working version of the product is built, in a series of stages, or iterations – hence the name iterative or incremental development. Each stage encompasses requirements definition, design, code and test. This is shown diagrammatically in Figure 2.3.

Figure 2.3 Iterative development



This type of development is often referred to as cyclical – we go 'round the development cycle a number of times', within the project. The project will have a defined timescale and cost. Within this, the cycles will be defined. Each cycle will also have a defined timescale and cost. The cycles are commonly referred to as time-boxes. For each time-box, a requirement is defined and a version of the code is produced, which will allow testing by the user representatives. At the end of each time-box, a decision is made on what extra functionality needs to be created for the next iteration. This process is then repeated until a fully working system has been produced.

A key feature of this type of development is the involvement of user representatives in the testing. Having the users represented throughout minimises the risk of developing an unsatisfactory product. The user representatives are empowered to request changes to the software, to meet their needs.

This approach to software development can pose problems, however.

The lack of formal documentation makes it difficult to test. To counter this, developers may use test-driven development. This is where functional tests are written first, and code is then created and tested. It is reworked until it passes the tests.

In addition, the working environment may be such that developers make any changes required, without formally recording them. This approach could mean that changes cannot be traced back to the requirements or to the parts of the software that have changed. Thus, traceability as the project progresses is reduced. To mitigate this, a robust process must be put in place at the start of the project to manage these changes (often part of a configuration management process – this is discussed further in Chapter 5).

Another issue associated with changes is the amount of testing required to ensure that implementation of the changes does not cause unintended changes to other parts of the software (this is called regression testing, discussed later in this chapter).

Forms of iterative development include prototyping, rapid application development (RAD) and agile software development. A proprietary methodology is the Rational Unified Process (RUP).

CHECK OF UNDERSTANDING

(1) What is meant by verification?
(2) What is meant by validation?
(3) Name three work-products typically shown in the V-model.
(4) Name three activities typically shown in the V-model.
(5) Identify a benefit of the V-model.
(6) Identify a drawback of the V-model.
(7) Name three activities typically associated with an iterative model.
(8) Identify a significant benefit of an iterative model.
(9) List three challenges of an iterative development.
(10) List three types of iterative development.
(11) Compare the work-products in the V-model with those in an iterative model.

TEST LEVELS

For both types of development, testing plays a significant role. Testing helps to ensure that the work-products are being developed in the right way (verification) and that the product will meet the user needs (validation).

Characteristics of good testing across the development life cycle include:

- Early test design – In the V-model, we saw that test planning begins with the specification documents. This activity is part of the fundamental test process discussed in Chapter 1. After test planning, the documents would be analysed and test cases designed. This approach would ensure that testing starts with the development of the requirements, i.e. a proactive approach to testing is undertaken. Proactive approaches to test design are discussed further in

Chapter 5. As we saw in iterative development, test-driven development may be adopted, pushing testing to the front of the development activity.

- Each work-product is tested – In the V-model, each document on the left is tested by an activity on the right. Each specification document is called the test basis, i.e. it is the basis on which tests are created. In iterative development, each release is tested before moving on to the next.
- Testers are involved in reviewing requirements before they are released – In the V-model, testers would be invited to review all documents from a testing perspective. Techniques for reviewing documents are outlined in Chapter 3.

In Figure 2.2, the test stages of the V-model are shown. They are often called test levels. The term test level provides an indication of the focus of the testing, and the types of problems it is likely to uncover. The typical levels of testing are:

- Unit (component) testing
- Integration testing
- System testing
- Acceptance testing

Each of these test levels will include tests designed to uncover problems specifically at that stage of development. These levels of testing can be applied to iterative development also. In addition, the levels may change depending on the system. For instance, if the system includes some software developed by external parties, or bought off the shelf, acceptance testing on these may be conducted before testing the system as a whole.

Let us now look at these levels of testing in more detail.

Unit (component) testing

Before testing of the code can start, clearly the code has to be written. This is shown at the bottom of the V-model. Generally, the code is written in component parts, or units. The units are usually constructed in isolation, for integration at a later stage. Units are also called programs, modules or components.

Unit testing is intended to ensure that the code written for the unit meets its specification, prior to its integration with other units.

In addition to checking conformance to the program specification, unit testing would also verify that all of the code that has been written for the unit can be executed. Instead of using the specification to decide on inputs and expected outputs, the developer would use the code that has been written for this. Testing based on code is discussed in detail in Chapter 4. Thus the test bases for unit testing can include: the component requirements; the detailed design; the code itself.

Unit testing requires access to the code being tested. Thus test objects (i.e. what is under test) can be the components, the programs, data conversion/migration programs and database modules. Unit testing is often supported by a unit

test framework (e.g. Kent Beck's Smalltalk Testing Framework: http://xprogramming.com/testfram.htm). In addition, debugging tools are often used.

An approach to unit testing is called Test Driven Development. As its name suggests, test cases are written first, code built, tested and changed until the unit passes its tests. This is an iterative approach to unit testing.

Unit testing is usually performed by the developer who wrote the code (and who may also have written the program specification). Defects found and fixed during unit testing are often not recorded.

Integration testing

Once the units have been written, the next stage would be to put them together to create the system. This is called integration. It involves building something large from a number of smaller pieces.

The purpose of integration testing is to expose defects in the interfaces and in the interactions between integrated components or systems.

Thus the test bases for integration testing can include: the software and system design; a diagram of the system architecture; workflows and use-cases.

The test objects would essentially be the interface code. This can include subsystems' database implementations.

Before integration testing can be planned, an integration strategy is required. This involves making decisions on how the system will be put together prior to testing. There are three commonly quoted integration strategies, as follows.

Big-bang integration

This is where all units are linked at once, resulting in a complete system. When testing of this system is conducted, it is difficult to isolate any errors found, because attention is not paid to verifying the interfaces across individual units.

This type of integration is generally regarded as a poor choice of integration strategy. It introduces the risk that problems may be discovered late in the project, where they are more expensive to fix.

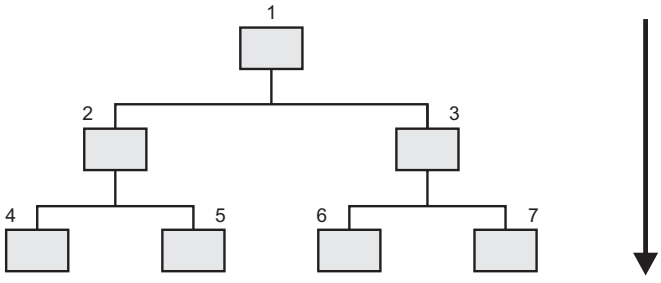
Top-down integration

This is where the system is built in stages, starting with components that call other components. Components that call others are usually placed above those that are called. Top-down integration testing will permit the tester to evaluate component interfaces, starting with those at the 'top'.

Let us look at the diagram in Figure 2.4 to explain this further.

The control structure of a program can be represented in a chart. In Figure 2.4, component 1 can call components 2 and 3. Thus in the structure, component 1 is placed above components 2 and 3. Component 2 can call components 4 and 5. Component 3 can call components 6 and 7. Thus in the structure, components 2 and 3 are placed above components 4 and 5 and components 6 and 7, respectively.

Figure 2.4 Top-down control structure



In this chart, the order of integration might be:

- 1,2
- 1,3
- 2,4
- 2,5
- 3,6
- 3,7

Top-down integration testing requires that the interactions of each component must be tested when it is built. Those lower down in the hierarchy may not have been built or integrated yet. In Figure 2.4, in order to test component 1's interaction with component 2, it may be necessary to replace component 2 with a substitute since component 2 may not have been integrated yet. This is done by creating a skeletal implementation of the component, called a stub. A stub is a passive component, called by other components. In this example, stubs may be used to replace components 4 and 5, when testing component 2.

The use of stubs is commonplace in top-down integration, replacing components not yet integrated.

Bottom-up integration

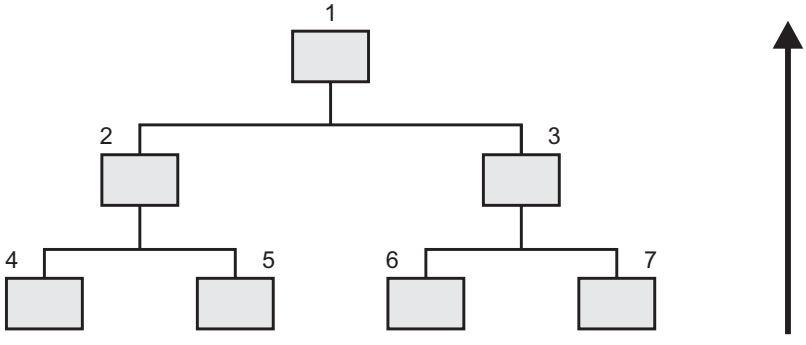
This is the opposite of top-down integration and the components are integrated in a bottom-up order. This is shown in Figure 2.5.

The integration order might be:

- 4,2
- 5,2
- 6,3
- 7,3

- 2,1
- 3,1

Figure 2.5 Bottom-up integration



So, in bottom-up integration, components 4–7 would be integrated before components 2 and 3. In this case, the components that may not be in place are those that actively call other components. As in top-down integration testing, they must be replaced by specially written components. When these special components call other components, they are called drivers. They are so called because, in the functioning program, they are active, controlling other components.

Components 2 and 3 could be replaced by drivers when testing components 4–7. They are generally more complex than stubs.

There may be more than one level of integration testing. For example:

- Component integration testing focuses on the interactions between software components and is done after component (unit) testing. This type of integration testing is usually carried out by developers.
- System integration testing focuses on the interactions between different systems and may be done after system testing of each individual system. For example, a trading system in an investment bank will interact with the stock exchange to get the latest prices for its stocks and shares on the international market. This type of integration testing is usually carried out by testers.

It should be noted that testing at system integration level carries extra elements of risk. These can include: at a technical level, cross-platform issues; at an operational level, business workflow issues; and at a business level, risks associated with ownership of regression issues associated with change in one system possibly having a knock-on effect on other systems.

System testing
Having checked that the components all work together at unit level, the next step is to consider the functionality from an end-to-end perspective. This activity is called system testing.

System testing is necessary because many of the criteria for test selection at unit and integration testing result in the production of a set of test cases that are unrepresentative of the operating conditions in the live environment. Thus testing at these levels is unlikely to reveal errors due to interactions across the whole system, or those due to environmental issues.

System testing serves to correct this imbalance by focusing on the behaviour of the whole system/product as defined by the scope of a development project or programme, in a representative live environment. It is usually carried out by a team that is independent of the development process. The benefit of this independence is that an objective assessment of the system can be made, based on the specifications as written, and not the code.

In the V-model, the behaviour required of the system is documented in the functional specification. It defines what must be built to meet the requirements of the system. The functional specification should contain definitions of both the functional and non-functional requirements of the system.

A functional requirement is a requirement that specifies a function that a system or system component must perform. Functional requirements can be specific to a system. For instance, you would expect to be able to search for flights on a travel agent's website, whereas you would visit your online bank to check that you have sufficient funds to pay for the flight.

Thus functional requirements provide detail on what the application being developed will do.

Non-functional system testing looks at those aspects that are important but not directly related to what functions the system performs. These tend to be generic requirements, which can be applied to many different systems. In the example above, you can expect that both systems will respond to your inputs in a reasonable time frame, for instance. Typically, these requirements will consider both normal operations and behaviour under exceptional circumstances.

Thus non-functional requirements detail how the application will perform in use.

Examples of non-functional requirements include:

- Installability – installation procedures.
- Maintainability – ability to introduce changes to the system.
- Performance – expected normal behaviour.
- Load handling – behaviour of the system under increasing load.

- Stress handling – behaviour at the upper limits of system capability.
- Portability – use on different operating platforms.
- Recovery – recovery procedures on failure.
- Reliability – ability of the software to perform its required functions over time.
- Usability – ease with which users can engage with the system.

Note that security and interoperability with specific systems are regarded as functional requirements in this syllabus.

The amount of testing required at system testing, however, can be influenced by the amount of testing carried out (if any) at the previous stages. In addition, the amount of testing advisable would also depend on the amount of verification carried out on the requirements (this is discussed further in Chapter 3).

Test bases for system testing can include: system and software requirement specifications; use cases; functional specifications; risk analysis reports; and system, user and operation manuals.

The test object will generally be the system under test.

Acceptance testing

The next step after system testing is often acceptance testing. The purpose of acceptance testing is to provide the end users with confidence that the system will function according to their expectations. Referring once more to the V-model, acceptance testing will be carried out using the requirement specification as a basis for test.

The requirement specification is typically the first document to be written, after initial capture of the user requirement. An example of a requirement could be to create a website that enables users to buy airline tickets online.

The subsequent documentation (functional, technical and program specifications) will expand on this in increasing levels of detail, in order to facilitate development of the system, as seen earlier. Thus, it is paramount that these requirements are fully documented and correct before further development activity is carried out. Again, this is the V-model approach. You may well be aware that having such an ideal set of requirements is a rare thing. This does not mean, however, that the need for correctness and completeness should be ignored. Techniques for verifying requirements are given in Chapter 3.

As for system testing, no reference is made to the code from which the system is constructed. Thus the test bases can include: user requirements; system requirements; use cases; business processes; and risk analysis reports.

The test objects can include: the fully integrated system; forms and reports produced by the system.

Unlike system testing, however, the testing conducted here should be independent of any other testing carried out. Its key purpose is to demonstrate system conformance to, for example, the customer requirements and operational and maintenance processes. For instance, acceptance testing may assess the system's readiness for deployment and use.

Acceptance testing is often the responsibility of the customers or users of a system, although other project team members may be involved as well.

Typical forms of acceptance testing include the following:

- User acceptance testing – testing by user representatives to check that the system meets their business needs. This can include factory acceptance testing, where the system is tested by the users before moving it to their own site. Site acceptance testing could then be performed by the users at their own site.
- Operational acceptance testing – often called operational readiness testing. This involves checking that the processes and procedures are in place to allow the system to be used and maintained. This can include checking:
    - Back-up facilities
    - Procedures for disaster recovery
    - Training for end users
    - Maintenance procedures
    - Data load and migration tasks
    - Security procedures
- Contract and regulation acceptance testing
    - Contract acceptance testing – sometimes the criteria for accepting a system are documented in a contract. Testing is then conducted to check that these criteria have been met, before the system is accepted.
    - Regulation acceptance testing – in some industries, systems must meet governmental, legal or safety standards. Examples of these are the defence, banking and pharmaceutical industries.
- Alpha and beta testing
    - Alpha testing takes place at the developer's site – the operational system is tested whilst still at the developer's site by internal staff, before release to external customers. Note that testing here is still independent of the development team.
    - Beta testing takes place at the customer's site – the operational system is tested by a group of customers, who use the product at their own locations and provide feedback, before the system is released. This is often called 'field testing'.

CHECK OF UNDERSTANDING

(1) In the V-model, which document would be used as the test basis for unit testing?

(2) Describe three typical integration strategies.

(3) Identify why stubs and drivers are usually used.

(4) In the V-model, which document is used as the test basis for system testing?

(5) Compare a functional requirement with a non-functional requirement.

(6) List three non-functional requirements.

(7) What is the purpose of acceptance testing?

(8) In the V-model, what is the test basis for acceptance testing?

(9) Identify three types of acceptance testing.

TEST TYPES

In the last section we saw that each test level has specific testing objectives. In this section we will look at the types of testing required to meet these objectives.

Test types fall into the following categories:

- Functional testing
- Non-functional testing
- Structural testing
- Testing after code has been changed.

To facilitate different types of testing, models may be used as follows:

- Functional testing: process flows; state transition models; security threat models; plain language specifications.
- Non-functional testing: performance model; usability model.
- Structural testing: control flow model; menu structure model.

Functional testing

As you saw in the section on system testing, functional testing looks at the specific functionality of a system, such as searching for flights on a website, or perhaps calculating employee pay correctly using a payroll system. Note that security testing is a functional test type. Another type of functional testing is interoperability testing – this evaluates the capability of the system to interact with other specified components.

Functional testing is also called specification-based testing: testing against a specification.

Non-functional testing
This is where the behavioural aspects of the system are tested. As you saw in the section on system testing, examples are usability, performance under load and stress, among others. As for functional testing, these requirements are usually documented in a functional specification. Thus, mainly black-box testing techniques are used for this type of testing.

These tests can be referenced against a quality model, such as the one defined in ISO 9126 *Software Engineering – Software Product Quality*. Note that a detailed understanding of this standard is not required for the exam.

Structural testing
This type of testing is used to measure how much testing has been carried out. In functional testing, this could be the number of functional requirements tested against the total number of requirements.

In structural testing, we change our measure to focus on the structural aspects of the system. This could be the code itself, or an architectural definition of the system. We want to do this to check the thoroughness of the testing carried out on the system that has actually been built. A common measure is to look at how much of the actual code that has been written has been tested. Further detail on code coverage measures is provided in Chapter 4.

Note that structural testing can be carried out at any test level.

Testing related to changes
The previous sections detail the testing to be carried out at the different stages in the development life cycle. At any level of testing, it can be expected that defects will be discovered. When these are found and fixed, the quality of the system being delivered can be improved.

After a defect is detected and fixed the changed software should be retested to confirm that the problem has been successfully removed. This is called retesting or confirmation testing. Note that when the developer removes the defect, this activity is called debugging, which is not a testing activity. Testing finds a defect, debugging fixes it.

The unchanged software should also be retested to ensure that no additional defects have been introduced as a result of changes to the software. This is called regression testing. Regression testing should also be carried out if the environment has changed.

Regression testing involves the creation of a set of tests which serve to demonstrate that the system works as expected. These would be run again many times over a testing project, when changes are made, as discussed above. This repetition of tests makes regression testing suitable for automation in many cases. Test automation is covered in detail in Chapter 6.

CHECK OF UNDERSTANDING

Which of the following is correct?

(a) Regression testing checks that a problem has been successfully addressed, whilst confirmation testing is done at the end of each release.

(b) Regression testing checks that all problems have been successfully addressed, whilst confirmation testing refers to testing individual fixes.

(c) Regression testing checks that fixes to errors do not introduce unexpected functionality into the system, whilst confirmation testing checks that fixes have been successful.

(d) Regression testing checks that all required testing has been carried out, whilst confirmation testing checks that each test is complete.

MAINTENANCE TESTING

For many projects (though not all) the system is eventually released into the live environment. Hopefully, once deployed, it will be in service as long as intended, perhaps for years or decades.

During this deployment, it may become necessary to change the system. Changes may be due to:

- Additional features being required.
- The system being migrated to a new operating platform.
- The system being retired – data may need to be migrated or archived.
- Planned upgrade to COTS-based systems.
- New faults being found requiring fixing (these can be 'hot fixes').

Once changes have been made to the system, they will need to be tested (retesting), and it also will be necessary to conduct regression testing to ensure that the rest of the system has not been adversely affected by the changes. Testing that takes place on a system which is in operation in the live environment is called maintenance testing.

When changes are made to migrate from one platform to another, the system should also be tested in its new environment. When migration includes data being transferred in from another application, then conversion testing also becomes necessary.

As we have suggested, all changes must be tested, and, ideally, all of the system should be subject to regression testing. In practice, this may not be feasible or cost-effective. An understanding of the parts of the system that could be affected by the changes could reduce the amount of regression testing required. Working this out is termed impact analysis, i.e. analysing the impact of the changes on the system.

Impact analysis can be difficult for a system that has already been released. This is because the specifications may be out of date (or non-existent), and/or the original development team may have moved on to other projects, or left the organisation altogether.

CHECK OF UNDERSTANDING

(1) What is the purpose of maintenance testing?

(2) Give examples of when maintenance testing would be necessary.

(3) What is meant by the term impact analysis?

SUMMARY

In this chapter we have explored the role of testing within the software development life cycle. We have looked at the basic steps in any development model, from understanding customer needs to delivery of the final product. These were built up into formally recognisable models, using distinct approaches to software development.

The V-model, as we have seen, is a stepwise approach to software development, meaning that each stage in the model must be completed before the next stage can be started, if a strict implementation of the model is required. This is often the case in safety-critical developments. The V-model typically has the following work-products and activities:

(1) Requirement specification

(2) Functional specification

(3) Technical specification

(4) Program specification

(5) Code

(6) Unit testing

(7) Integration testing

(8) System testing

(9) Acceptance testing

Work-products 1–5 would be subject to verification, to ensure that they have been created following the rules set out. For example, the program specification would be assessed to ensure that it meets the requirements set out in the technical specification, and that it contains sufficient detail for the code to be produced.

In activities 6–9, the code is assessed progressively for compliance to user needs, as captured in the specifications for each level.

An iterative model for development has fewer steps, but involves the user from the start. These steps are typically:

(1) Define iteration requirement.

(2) Build iteration.

(3) Test iteration.

This sequence would be repeated for each iteration until an acceptable product has been developed.

An explanation of each of the test levels in the V-model was given. For unit testing the focus is the code within the unit itself, for integration testing it is the interfacing between units, for system testing it is the end-to-end functionality, and for acceptance testing it is the user perspective.

An explanation of test types was then given and by combining alternative test types with test levels we can construct a test approach that matches a given system and a given set of test objectives very closely. The techniques associated with test types are covered in detail in Chapter 4 and the creation of a test approach is covered in Chapter 5.

Finally, we looked at the testing required when a system has been released, but a change has become necessary – maintenance testing. We discussed the need for impact analysis in deciding how much regression testing to do after the changes have been implemented. This can pose an added challenge, if the requirements associated with the system are missing or have been poorly defined.

In the next chapter, techniques for improving requirements will be discussed.

Example examination questions with answers

E1. K1 question

Which of the following is usually the test basis for integration testing?

a. Program specification
b. Functional specification
c. Technical specification
d. Requirement specification

E2. K2 question

A top-down development strategy affects which level of testing most?

a. Component testing
b. Integration testing
c. System testing
d. User acceptance testing

E3. K2 question

Which of the following is a non-functional requirement?

a. The system will enable users to buy books.
b. The system will allow users to return books.
c. The system will ensure security of the customer details.
d. The system will allow up to 100 users to log in at the same time.

E4. K1 question

Which of the following are examples of iterative development models?

(i) V-model
(ii) Rapid Application Development model
(iii) Waterfall model
(iv) Agile development model

a. (i) and (ii)
b. (ii) and (iii)
c. (ii) and (iv)
d. (iii) and (iv)

E5. K2 question

Which of the following statements are true?

(i) For every development activity there is a corresponding testing activity.
(ii) Each test level has the same test objectives.
(iii) The analysis and design of tests for a given test level should begin after the corresponding development activity.
(iv) Testers should be involved in reviewing documents as soon as drafts are available in the development life cycle.

a. (i) and (ii)
b. (iii) and (iv)
c. (ii) and (iii)
d. (i) and (iv)

E6. K1 question
Which of the following is not true of regression testing?

a. It can be carried out at each stage of the life cycle.
b. It serves to demonstrate that the changed software works as intended.
c. It serves to demonstrate that software has not been unintentionally changed.
d. It is often automated.

Answers to questions in the chapter

SA1. The correct answer is d.
SA2. The correct answer is b.
SA3. The correct answer is a.

Answers to example questions

E1. The correct answer is c.
Option (a) is used for unit testing. Option (b) is used for system testing and option (d) is used for acceptance testing.

E2. The correct answer is b.
The development strategy will affect the component testing (option (a)), in so far as it cannot be tested unless it has been built. Options (c) and (d) require the system to have been delivered; at these points the development strategy followed is not important to the tester. Option (b) needs knowledge of the development strategy in order to determine the order in which components will be integrated and tested.

E3. The correct answer is d.
The other options are functional requirements. Note that security is regarded as a functional requirement in this syllabus.

E4. The correct answer is c.
The other two models are sequential models.

E5. The correct answer is d.
Option (ii) is incorrect: each test level has a different objective. Option (iii) is also incorrect: test analysis and design should start once the documentation has been completed.

E6. The correct answer is b.
This is a definition of confirmation testing. The other three options are true of regression testing.